NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DEVELOPING A LIBRARY FOR PROOFS OF DATA POSSESSION IN CHARM

by

Krisztina C. Riebel-Charity

June 2013

Thesis Advisor: Mark Gondree
Co-Advisor: Zachary Peterson

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 2013	Master's Thesis

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
DEVELOPING A LIBRARY FOR PROOFS OF DATA POSSESSION IN CHARM	
6. AUTHOR(S) Krisztina C. Riebel-Charity	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943–5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited	

13. ABSTRACT (maximum 200 words)

Provable Data Possession (PDP) is a cryptographic tool for auditing big data on a storage server or in the cloud. The goal of PDP is to efficiently verify that the server is storing the data. PDP provides probabilistic guarantees that the server is storing the information, and it will be available when needed, without accessing the entire file.

In this work, we have developed a PDP module for the Charm cryptographic framework. We wrote an application programmer interface (API) for generic PDP schemes. We implemented the simple MAC-PDP scheme with efficient subroutines for sub-linear sampling. We hope that this work will encourage further study in the rapid prototyping and evaluation of new PDP schemes in the Charm framework.

14. SUBJECT TERMS API, Charm, MAC-PDPD, PDP, POR	15. NUMBER OF PAGES 43
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540–01–280–5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DEVELOPING A LIBRARY FOR PROOFS OF DATA POSSESSION IN CHARM

Krisztina C. Riebel-Charity
Civilian, Department of the Navy
B.S., Babes-Bolyai University, Cluj-Napoca, Romania, July 2010

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2013

Author: Krisztina C. Riebel-Charity

Approved by: Mark Gondree
Thesis Advisor

Zachary Peterson
Thesis Co-Advisor

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Provable Data Possession (PDP) is a cryptographic tool for auditing big data on a storage server or in the cloud. The goal of PDP is to efficiently verify that the server is storing the data. PDP provides probabilistic guarantees that the server is storing the information, and it will be available when needed, without accessing the entire file.

In this work, we have developed a PDP module for the Charm cryptographic framework. We wrote an application programmer interface (API) for generic PDP schemes. We implemented the simple MAC-PDP scheme with efficient subroutines for sub-linear sampling. We hope that this work will encourage further study in the rapid prototyping and evaluation of new PDP schemes in the Charm framework.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Programmer Interface
DTP	Datatype-preserving Encryption
FedRAMP	U.S. Federal Risk and Authorization Management Program
MAC	Message Authentication Code
MAC-PDP	Message Authentication Code PDP
PDP	Provable Data Possession
POR	Proof of Retrievability
SLA	Service Level Agreement

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Partial support for this work was provided by the National Science Foundation's CyberCorps: Scholarship for Service (SFS) program under Award No. 0912048. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

I would like to express my gratitude and appreciation to my advisor, Professor Mark Gondree, for his direction and guidance in helping me to develop and write my thesis. Without Professor Gondree’s tireless support, I would not have been able to complete this work. I would also like to thank Mrs. Diana Chung for her encouragement and reassurance throughout the process of writing my thesis.

Finally, I would like to thank my husband, Bart, for his support and understanding throughout this process.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Provable Data Possession (PDP) is a cryptographic tool for auditing remote data, held on a storage server or in the cloud. The goal of PDP is to allow a client to efficiently verify the data on the server, without retrieving the entire file (as required for traditional integrity mechanisms, like digital signatures). This ability makes PDP audits particularly efficient for verifying the integrity of big data, for which accessing the entire file is expensive in terms of bandwidth and time. PDP provides a probabilistic (rather than absolute) guarantee that the server is storing the information and it will be accessible when needed. The U.S. Federal Cloud Computing Strategy recommends vendors be held accountable for service failures, using active Service Level Agreement (SLA) compliance monitoring [1]. Likewise, the U.S. Federal Risk and Authorization Management Program (FedRAMP) mandates continuous, active monitoring of services [2]. PDP protocols provide such a mechanism for actively auditing outsourced storage. This is particularly relevant to the Intelligence Community and Department of Defense, both of which face big data management problems. For example, the National Geospatial-Intelligence Agency anticipates collecting on the order of four petabytes, annually [3]. We believe PDP can be a "game changing" technology for securing big data in the context of cloud technologies.

Efficiency is integral to PDP audits; however, an in-depth cost comparison of PDP schemes is largely absent from the literature. Existing cost analyses consist of asymptotic costs measuring communication and round complexity, and small-scale experimental validation using proof-of-concept code (which varies from study to study and is unpublished). Our work is preparatory, allowing others to provide a more thorough analysis and to contribute to a shared "ecosystem" of experimental PDP implementations. We implemented a PDP module for the Charm cryptographic framework, a Python-based library for rapid cryptographic prototyping and benchmarking. We have developed a PDP application programmer interface (API) with documentation, implemented the simple Message Authentication Code PDP (MAC-PDP) scheme with unit tests, and provided efficient subroutines for sub-linear sampling in a cryptographic setting based on Floyd's

algorithm. We hope these tools will facilitate further study in PDP performance experimentation and allow the rapid prototyping and evaluation of novel PDP schemes in Charm.

II. BACKGROUND

The objective of this research is to develop a set of tools for the CHARM framework that enable studying PDP, a cryptographic tool for auditing big data. Our initial work implements the MAC-PDP scheme, a simple PDP scheme based on message authentication codes (MACs) to which many novel PDP schemes have been compared [4]. Here, we provide an overview of PDP schemes, the MAC-PDP scheme and the CHARM cryptographic framework.

A. PDP

PDP is a cryptographic protocol that enables a client to audit data on a remote location without accessing the whole file. When verifying the authenticity of the information stored on the server, strategies based on replication incur high cost and provide low integrity in the face of adversaries. Cryptographic signatures provide suitable integrity guarantees, but require the entire file be retrieved. With PDP we are able to get a *probabilistic* guarantee for integrity, at significant cost savings. Most PDP schemes have the property that its audit's probabilistic guarantee can be exponentially amplified through repetition. Juels and Kaliski [4] and Naor and Rothblum [5] were the first to define these type of protocols. They describe different but related approaches to verify the authenticity of remote data: Provable Data Possession (PDP) and Proof of Retrievability (POR). Both approaches pre-process a file by splitting it into blocks, store the file remotely and then verify the file's integrity using an interactive challenge-response protocol. POR schemes have the additional property that this challenge-response protocol admits an "extractor," so that an efficient algorithm can be used to reconstruct the original file (i.e., if a server can consistently pass POR audits then the original file, in its entirety, can be recovered by the client). Their POR scheme uses special blocks called sentinels, hidden between other blocks in the data [6]. A number of PDP and POR schemes have since been proposed in the literature [6]–[13]. We focus on PDP schemes in this work but note the deep connections between PDP and POR schemes.

In general, a PDP scheme can be constructed in a few phases: (a) secret generation, (b) pre-processing or tagging and (c) an interactive proof phase. In the tagging phase the user pre-processes the file, generating some *tag data* and transmits this to the server with the file. The server stores the file and tag data, while the user stores only the key. In the proof phase, the user generates and sends a challenge to the server, the server responds to the challenge with a short proof, and the user verifies this proof.

B. MAC-PDP

MAC-PDP is a simple PDP scheme based on symmetric key cryptography. The idea is to break a file into blocks, create a short tag for each block using a message authentication code (MAC), and send both to the remote server. Then, a client can randomly sample blocks and their corresponding tags, verifying the integrity of each block. If this audit is passed then, with a high probability, portions of the file have not been lost, corrupted or tampered. We describe the phases and arguments for MAC-PDP, next.

Protocol Phases. In the key generation phase of MAC-PDP, one chooses a random secret key, $\mathbf{sk} = \langle k_m \rangle$, that will be used to MAC each file block. In the pre-processing phase, we split up the file *M* into *n* data blocks, each of length *s*. We choose a unique file name ID_M for the file *M*. For each block *b*, we generate a MAC tag $\sigma_b = \mathrm{MAC}(k_m, \langle ID_M \| b \rangle)$. We can store all of this data—unique filename, blocks and their tags—on the remote server. In the interactive proof phase, the client selects *c* random block indices and sends this to the server as a challenge. The server returns the *c* blocks and tags corresponding to those indices, as a proof. Finally, the client uses its secret key to verify the proof, re-computing each MAC and comparing to the stored tag. For a secure MAC, it is computationally infeasible for a malicious server to forge valid tag data, i.e., by inventing a new tag for an existing block or for totally inventing a new block/tag pair. We direct the interested reader to the security proof provided by Shacham and Waters for the MAC-PDP scheme, for details on these arguments [14].

Complexity. The communication complexity for the client during the interactive proof phase is $O(c\log(n))$. The communication complexity for the server during the interactive proof phase is $O(c(s+\sigma))$, where σ is the length of the MAC digest.

Parameter Selection. MAC-PDP gives a probabilistic guarantee that the remote server is storing authentic data. Let P_X be the probability of the event that *t* blocks have deleted or corrupted, and that an audit of *c* random blocks detects one or more of these blocks. Ateniese et al. observe this probability is bound by:

$$1-\left(\frac{n-c}{n}\right)^c \le P_X \le 1-\left(\frac{n-c+1-t}{n-c+1}\right)^c$$

Further, they observe that if t is more than 1% of the total number of blocks, this can be detected (with a probability of 99%) if the client challenges 460 data blocks [7].

C. CHARM FRAMEWORK

Charm is a framework for rapidly prototyping and benchmarking experimental cryptosystems. Charm is based on the Python programming language and it was designed to reduce code complexity by promoting reuse of cryptographic components within the framework [15]. Charm contains four "Base Modules" that implement various core cryptographic routines, written in C and optimized for efficiency: the crypto module (blockciphers), benchmark module, math module (big numbers, elliptic curve operations, pairing-based operations), and utilities module (base64 encoding and hash functions). Charm also contains a "Toolbox Module" that contains Python APIs exposing some of these cryptographic primitives, and a "Scheme Module" for various cryptographic schemes (e.g, Schnorr's zero-knowledge proof system, and various public key signature schemes). It is important to note that Charm is not a library providing advanced cryptographic functions for production-ready systems: it ignores issues like standards compliance, secure coding practices and appropriate key management. It is a tool for cryptographic research and study.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DESIGN

We designed a generic PDP API for Charm by developing those interfaces any future PDP scheme would instantiate. Charm already supports a generic interactive protocol class, which our PDP base class extends. We were able to reuse many of the components already provided by Charm's protocol engine: the client/server logic to establish the connection between the PDP challenger and the PDP prover, data serialization logic, and the logic defining protocol states and how to perform state transitions. We provide an overview of the PDP class interfaces, and the generic PDP protocol transitions. We refer to the local client that stores the file data as the "Challenger" and the remote data server which responds to audits as the "Prover." Of course, some PDP schemes allow public audits in which the Challenger does not need to be the same party that initially stores the file. Without loss of generality, it is appropriate to assume these parties are the same for the purposes of rapid prototyping and measurement in Charm. Any production library for a PDP scheme offering public verifiability would likely make alternative design choices.

A. PDP INTERFACES

The PDP base scheme API consists of the following five interfaces:

keyGen: Generates keys for use in the scheme.

tag: Generates tag data to be stored with the Prover and tag data to be stored by the Challenger. Returns tag data related to the filename.

generateChallenge: Generates a challenge that can be sent to a Prover.

generateProof: Generates a proof, responding to a challenge.

verifyProof: Checks to see if a proof is valid relative to its challenge.

B. PDP PROTOCOL STATE TRANSITIONS

Our PDP base class includes a generic three-round challenge-response protocol making use of the previous interfaces. We acknowledge that some PDP protocols may

have a different round structure. For those schemes, instead of inheriting the PDP base class protocol logic, an implementer would over-ride these routines in the child class. However, most existing PDP schemes are three-round interactive proofs, and will be able to inherit and re-use this protocol structure directly. The details of each of these states (see Figure 1) are outlined below.

Challenger State 1: This is the initial state for the Challenger.

- Input: None
- Processing: Generate public and private key data; pre-process the local file to create the tag data; sends the file and tag data to the remote server.
- Sends M1: The file and its tag data.
- Stores: Public and private key data and, if applicable, any private file-specific data (used during verification).

Prover State 2: The Prover enters this state upon receiving the message from State 1.

- Input: Data from the Challenger, to be stored.
- Processing: Store data and send acknowledgement.
- Sends M2: An acknowledgement of success.
- Stores: The file and its tag data.

Challenger State 3: The Challenger issues a challenge to the Prover.

- Input: Acknowledgement that the Prover is ready for an audit.
- Processing: A challenge is generated and sent to the Prover.
- Sends M3: A challenge.
- Stores: The challenge, to be used during the verification stage.

Prover State 4: The Prover generates a proof and sends it to the Challenger.

- Input: A challenge.
- Processing: The Prover generates a proof.
- Sends M4: A proof.
- Stores: None.

Challenger State 5: The Challenger verifies the proof.

- Input: A proof.

- Processing: The Challenger checks if the proof is valid. If the verification fails, go to the Fail state; if success and the maximum number of challenges have been issued, go to the Success state; otherwise, return to the state to issue another challenge.
- Sends: The result of the audit.
- Stores: The number of successful or failed audits.

Prover State 6: The Prover receives the result from the Challenger.

- Input: A status message indicating the result.
- Processing: If the proof was verified and the maximum number of challenges has not been issued, return to respond to a new challenge; if the proof failed verification, go to the Fail state; otherwise, go to the Success state.
- Sends: None.
- Stores: None.

Challenger State 7: This is the Challenger Fail state.

- Input: None.
- Processing: None. The Challenger reaches this state after a proof fails an audit.
- Sends: None.
- Stores: None.

Prover State 8: This is the Prover Fail state.

- Input: None.
- Processing: None. If the Prover reaches this state after any proof fails the audit.
- Sends: None.
- Stores: None.

Challenger State 9: This is the Challenger Success state.

- Input: None.
- Processing: None. The Challenger reaches this state after all proofs pass the audit.
- Sends: None.
- Stores: None.

Prover State 10: This is the Prover Success state.

- Input: None.
- Processing: None. The Prover reaches this state after all proofs pass the audit**.**
- Sends: None.
- Stores: None.

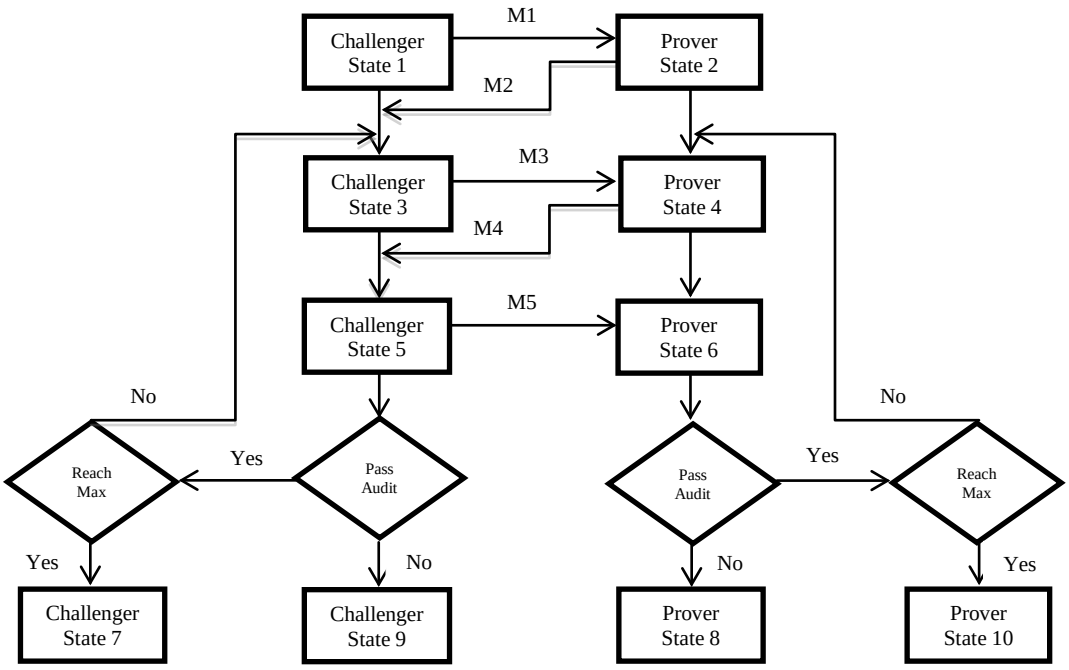

Figure 1. Data Flow Diagram

C. CONCLUSION

We have described the core interfaces provided by the PDP base class, which our MAC-PDP scheme implements. Next, we describe the MAC-PDP scheme implementation and some notable changes and support functions developed for this task.

IV. IMPLEMENTATION

We discuss our implementation of the MAC-PDP scheme, which is a concrete instantiation of the previous interfaces.

A. MAC-PDP

MAC-PDP is a simple PDP scheme based on symmetric key cryptography. We implemented the following interfaces for the MAC-PDP scheme.

keygen: Generates a symmetric key used to MAC blocks.

- Input: None
- Processing: Generate a private key using the *randomBits* function, provided by Charm's *integer* module.
- Output: The private key *sk*. The public key *pk* is empty.

tag: Generates the tag data to be sent to the storage server.

- Input: The name of the file to tag, and the key pair (*sk*, *pk*).
- Processing: It breaks the file into blocks. It creates a unique filename, by appending a unique index to the filename. It creates a plaintext message by concatenating the unique filename, the block-number, and the block of data. A MAC for the plaintext is generated using Charm's *MessageAuthenticator* class.
- Output: The *remotedata* is a list of tags. The *localdata* is the number of blocks tagged.

generateChallenge: Generates a list of *m* indices chosen randomly in the range [0, r), where *r* is the number of blocks, a value stored in *localdata*. The value *m* is a system parameter, set during the initialization routine.

- Input: The *localdata*, holds the parameter *r*.
- Processing: We select m indices in the range [0, *r*) without replacement, using Floyd's Algorithm.
- Output: The list of challenge indices.

generateProof: This function returns the tags and blocks associated with the indices in the challenge.

- Input: The *challenge*, *remotedata* and public key *pk* (empty).

- Processing: In this implementation, the tag data and file blocks are held in memory, in an array. Selecting the challenged blocks and tags is an array lookup. This implementation is not appropriate files that are too large to be held in-memory.
- Output: The *proof*, an array of blocks and tags, in the order requested by the challenge.

verifyProof: This algorithm verifies the proof sent in response to our challenge.

- Input: The *proof*, *challenge*, and key pair (*sk, pk*).
- Processing: We use the *verify* function of the *MessageAuthenticator* class to re-compute the MAC for each block, and compare with the corresponding tag.
- Stores: True, if passed verification, False otherwise.

B. IMPLEMENTATION DECISIONS

Charm is a framework for prototyping cryptosystems and benchmarking individual cryptographic operations. We do not implement persistent storage and logic for handling files too large to be held in memory. Additionally, we do not implement logic for secure key storage or data serialization. We consider these tasks appropriate for production-worthy libraries, which is not the goal of Charm or in the scope of our work.

In Charm, The *Protocol* class implements send and receive logic using a fixed-size buffer. Even for our relatively small files, this buffer was too small to transfer the data during our tag phase. We modified the class to transmit data too large for the existing fixed size buffer.

C. RANDOM SUBLINEAR SAMPLING

When generating a challenge, we select a random subset of the block indices. This logic is implemented using Floyd's algorithm to select random numbers without replacement. Floyd's is an efficient algorithm and relatively simple to implement [16], yet no (or few) cryptographic libraries provide an interface for this functionality. The statistical software SAS provides the Simple Random Sampling algorithm, implemented using Floyd's algorithm [17]. The software Clustal Omega—a program for aligning

protein sequences, developed by the Conway Institute UCD Dublin and founded by the Science Foundation Ireland—also implements[1] Floyd's algorithm [18].

An alternative to Floyd's algorithm is Knuth's Algorithm S. Algorithm S is simpler in design, and it is easier to understand its correctness; it operates according to following: to select *n* elements from the set [0, *M*] we randomly select a number less than *M* and add this to our output if it has not previously been selected [16]. This algorithm is not as efficient as Floyd's algorithm: in Floyd's, we add a number to our list on every iteration.

Another alternative to selecting *n* random numbers from a set *M* without replacement is the Knuth Shuffle, also known as the Fisher-Yates Shuffle. This method permutes the elements from the set [0, *M*] and chooses the first *n* numbers from the shuffled set. A proposed block-cipher mode for Datatype-Preserving Encryption (DTP) uses the Knuth Shuffle in one of its steps [19]. It may be advantageous to consider more efficient shuffling algorithms, like Floyd's, in these applications.

D. UNIT TESTING

We developed unit tests using Python's existing framework for automated testing. These tests run without user input, to implement a set of regression tests that can be used to ensure no library functionality becomes broken in the future.

1 Floyd's is implemented as the function RandomUniqueIntArray, found in src/clustal/util.c.

THIS PAGE INTENTIONALLY LEFT BLANK

V. FUTURE WORK

The Charm framework does not implement benchmarking for symmetric key crypto schemes. MAC-PDP is a scheme based on symmetric key cryptography. In order to measure the cost of MAC-PDP, benchmarking for symmetric key cryptography needs to be integrated into the charm framework.

We developed a PDP interface and implemented MAC-PDP scheme as part of the Charm framework. Follow-on work might implement other PDP schemes described in the literature [6]–[13] in the Charm framework.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

PDP is a cryptographic tool for auditing big data in the cloud. In this work we have developed a PDP module for the Charm cryptographic framework. We wrote an API for PDP schemes we believe is generic enough to describe all known PDP schemes. We implemented the simple MAC-PDP scheme with efficient subroutines for sub-linear sampling. We hope that this work will enable the rapid prototyping and evaluation of new PDP schemes in the Charm framework.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. PDPBASE

Author: Krisztina

Date: 05/04/2013

PDP is a cryptographic process that enables a client to audit data on a remote location without accessing the whole file.

class `PDPBase.PDPBase`(*common_input=None*)

Bases: `charm.core.engine.protocol.Protocol`

`challenger_state1`()

This is the initial state for the Challenger.

- Input: None
- Processing: Generate public and private key data; pre-process the local file to create the tag data; sends the file and tag data to the remote server.
- Message, M1: The file and its tag data.
- Stores: Public and private key data and, if applicable, any private file-specific data (used during verification).

`challenger_state3`(*input*)

The Challenger issues a challenge to the Prover.

- Input: Acknowledgement that the Prover is ready for an audit.
- Processing: A challenge is generated and sent to the Prover.
- Message, M3: A challenge.
- Stores: The challenge, to be used during the verification stage.

`challenger_state5`(*input*)

The Challenger verified the proof.

- Input: A proof.
- Processing: The Challenger checks if the proof if valid. If the verification failed, go to the Fail state; if success and the maximum number of challenges has been issued, go to the Success state; otherwise, return to the state to issue another challenge.
- Message: The result of the audit.
- Stores: The number of successful or failed audits.

`challenger_state7`(*input*)

This is the Challenger Fail state.

- Input: None.

- Processing: None. The Challenger reaches this state after a proof fails an audit.
- Message: None.
- Stores: None.

`challenger_state9`(*input*)
This is the Challenger Success state.

- Input: None.
- Processing: None. The Challenger reaches this state after all proofs pass the audit.
- Message: None.
- Stores: None.

`generateChallenge`(*filestate, pk, sk*)
Generates a challenge c, that can be sent to a Prover.

`generateProof`(*challenge, pk*)
Generates a proof p, based on responding to a challenge.

`keygen`()
Generates keys for use in the scheme.

`prover_state10`(*input*)
This is the prover Success state.

- Input: None.
- Processing: None. The Prover reaches this state after all proofs pass the audit.
- Message: None.
- Stores: None.

`prover_state2`(*input*)
The prover enters this state upon receiving the message from State 1.

- Input: Data from the Challenger, to be stored.
- Processing: Store data and send acknowledgement.
- Message, M2: An acknowledgement of success.
- Stores: The file and its tag data.

`prover_state4`(*input*)
The prover generates a proof and sends it to the Challenger.

- Input: A challenge.

- Processing: The Prover generates a proof.
- Message, M4: A proof.
- Stores: None.

`prover_state6`(*input*)
The prover receives the result from the Challenger.

- Input: A status message indicating the result.
- Processing: If the proof was verified and the maximum number of challenges has not been issued, return to respond to a new challenge; if the proof failed verification, go to the Fail state; otherwise, go to the Success state.
- Message: None.
- Stores: None.

`prover_state8`(*input*)
This is the Prover Fail state.

- Input: None.
- Processing: None. If the Prover reaches this state after any proof fails the audit.
- Message: None.
- Stores: None.

`set_attributes`(***kwargs*)
Optional: sets various non-default properties for the PDP scheme.

`start_service`(*options*)
Sets up scheme, to act as either a Challenger or Prover service.

`tag`(*filename, pk, sk*)
Generates tag data to be stored with the file, (remote-data) and tag data to be stored by the Prover (local-data). Returns tagdata related to the filename.

`verifyProof`(*proof, challenge, pk, sk*)
Checks to see if a proof is valid relative to a challenge.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. MACPDPSCHEME

Author: Krisztina

Date: 05/04/2013

MAC-PDP is a simple PDP scheme based on symmetric key cryptography. We implemented the following interfaces for the MAC-PDP scheme.

class `macpdpscheme.macpdpscheme`(*common_input=None*)

Bases: `charm.toolbox.PDPBase.PDPBase`

`generateChallenge`(*localdata*, *pk*, *sk*)

Generates a list of m indices chosen randomly in the range [0, r), where r is the number of blocks in *localdata*. The value *m* is a system parameter, set during the initialization routine.

- Input: The *localdata*, holding the parameter r.
- Processing: We select m indices in the range [0, r) without replacement, using Floyd's Algorithm.
- Output: The list of challenge indices.

`generateProof`(*challenge*, *pk*, *remotedata*)

This function returns the tags and blocks associated with the indices in the challenge.

- Input: The *challenge*, *remotedata* and public key *pk* (empty).
- Processing: In this implementation, the tag data and file blocks are held in memory, in an array. Selecting the challenged blocks and tags is an array lookup. This implementation is not appropriate files that are too large to be held in-memory.
- Output: The *proof*, an array of blocks and tags, in the order requested by the challenge.

`keygen`()

Generates a symmetric key used to MAC blocks.

- Input: None
- Processing: Generate a private key using the *randomBits* function, provided by Charm's *integer* module.
- Output: The private key *sk*. The public key *pk* is empty.

`set_attributes`(***kwargs*)

Sets optional attributes chosen by the user. Such attributes are:

- key length
- block size

- number of challenges

`tag`(*filename, pk, sk*)
Generates the tag data to be sent to the storage server.

- Input: The name of the file to tag, and the key pair (*sk, pk*).
- Processing: It breaks the file into blocks. It creates a unique filename, by appending a unique index to the filename. It creates a plaintext message by concatenating the unique filename, the block-number, and the block of data. A MAC for the plaintext is generated using Charm's *MessageAuthenticator* class.
- Output: The *remotedata* is a list of tags. The *localdata* is the number of blocks tagged.

`verifyProof`(*proof, challenge, pk, sk*)
This algorithm verifies the proof sent in response to our challenge.

- Input: The *proof, challenge*, and key pair (*sk, pk*).
- Processing: We use the *verify* function of the *MessageAuthenticator* class to re-compute the MAC for each block, and compare with the corresponding tag.
- Stores: True, if passed verification, False otherwise.

LIST OF REFERENCES

[1] V. Kundra, "Federal cloud computing strategy," 2011. Available: http://www.whitehouse.gov/sites/default/files/omb/assets/egov_docs/vivek-kundra-federal-cloud-computing-strategy-02142011.pdf

[2] C. Council, "Proposed security assessment & authorization for U.S. government cloud computing," 2010. Available: http://educationnewyork.com/files/Proposed-Security-Assessment-and-Authorization-for-Cloud-Computing.pdf

[3] P. Buxbaum, "GEOIN's big data challenge," *Geospatial Intelligence Forum: The Magazine of the National Intelligence Community*, vol. 10, no. 3, pp.4–7, April 2012. Available: http://goo.gl/sbYch

[4] A. Juels and B. S. Kaliski Jr, "PORs: Proofs of retrievability for large files," in *Proc. of the 14th ACM Conf. on Comput. and Commun. Security*, Alexandria, VA, 2007, pp. 584–597.

[5] M. Naor and G. N. Rothblum, "The complexity of online memory checking,"in *Found. of Comput. Sci., 2005. FOCS 2005. 46th Annu. IEEE Symp.*, Pittsburg, PA, 2005, pp. 573–582.

[6] G. Ateniese, R. Di Pietro, L. V. Mancini and G. Tsudik, "Scalable and efficient provable data possession," in *Proc. of the 4th Int. Conf. on Security and Privacy in Commun. Networks*, Istanbul, Turkey, 2008, p. 9.

[7] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson and D. Song, "Provable data possession at untrusted stores," in *Proc. of the 14th ACM Conf. on Comput. and Commun. Security*, Alexandria, VA, 2007, pp. 598–609.

[8] C. Wang, Q. Wang, K. Ren and W. Lou, "Privacy-preserving public auditing for data storage security in cloud computing," in *INFOCOM, 2010 Proc. IEEE*, San Diego, CA, 2010, pp. 1–9.

[9] Y. Zhu, H. Wang, Z. Hu, G. Ahn, H. Hu and S. S. Yau, "Efficient provable data possession for hybrid clouds," in *Proc. of the 17th ACM Conf. on Comput. and Commun. Security*, Chicago, IL, 2010, pp. 756–758.

[10] C. Erway, A. Küpçü, C. Papamanthou and R. Tamassia, "Dynamic provable data possession," in *Proc. of the 17th ACM Conf. on Comput. and Commun. Security*, Beijing, China, 2009, pp. 213–222.

[11] B. Chen and R. Curtmola, "Robust dynamic provable data possession," in *Distributed Comput. Syst. Workshops (ICDCSW), 32nd Int. Conf.*, Macau, China, 2012, pp. 515–525.

[12] R. Curtmola, O. Khan, R. Burns and G. Ateniese, “MR-PDP: Multiple-replica provable data possession,” in *Distributed Computing Syst. (ICDCS), 28th Int. Conf.*, Beijing, China, 2008, pp. 411–420.

[13] B. Purushothama and B. Amberker, “Publicly auditable provable data possession scheme for outsourced data in the public cloud using polynomial interpolation,” in *Recent Trends in Comput. Networks and Distributed Syst. Security* Springer, 2012, pp. 11–22.

[14] H. Shacham and B. Waters, “Compact proofs of retrievability,” in *Advances in Cryptology-ASIACRYPT 2008.* NY:Springer, 2008, pp. 90–107.

[15] J. A. Akinyele, C. Garman, I. Miers, M. W. Pagano, M. Rushanan, M. Green and A. D. Rubin, “Charm: A framework for rapidly prototyping cryptosystems,” *J. of Cryptographic Eng.*, pp. 1–18, 2011.

[16] J. Bentley and B. Floyd, “Programming pearls: a sample of brilliance,” *Commun. ACM*, vol. 30, pp. 754–757, 1987.

[17] SAS Institute Inc., “The SURVEYSELECT Procedure.” in *SAS/STAT 9.2 User’s Guide,* Cary, NC: SAS Publishing, 2008, p 376.

[18] *src/clustal/util.c File Reference*. Aug. 31, 2012. Available: http://www.clustal.org/omega/clustalo-api/util_8c.html.

[19] U. T. Mattsson, “Format-controlling encryption using datatype-preserving encryption.” Available: http:// csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/fcem/fcem-spec.pdf.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California